NOVA SCHOOL OF SCIENCE & TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

Synchronization

lecture 11 (2021-04-26)

Master in Computer Science and Engineering

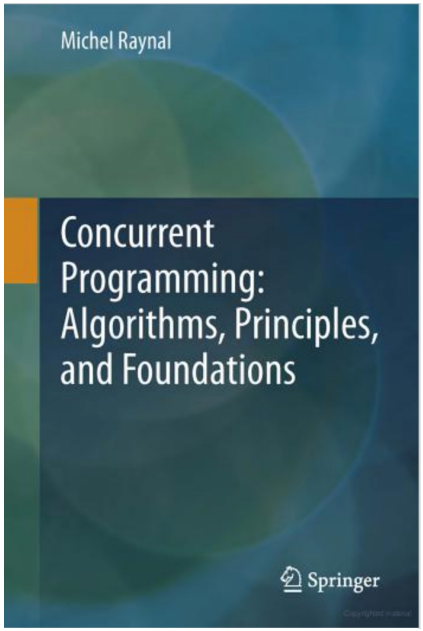
— Concurrency and Parallelism / 2020-21 —

João Lourenço <joao.lourenco@fct.unl.pt>

Summary

- **Synchronization**
  - Competition and Cooperation
  - Properties
  - Invariants

- **Reading list:**
  - **Chapter 1** of the book:
    Raynal M.;
    Concurrent Programming: Algorithms, Principles, and Foundations;
    Springer-Verlag Berlin Heidelberg (2013);
    ISBN: 978-3-642-32026-2



Algorithms, Programs and Processes

- A **sequential algorithm** is a formal description of the behavior of *a sequential state machine*
  - The text of the algorithm states the transitions that must be sequentially executed
- When an algorithm written in a specific programming language, it is called a **program**
- A **process** is an instance of a program
  - Hence an instance of an algorithm

Multiprocess Programs

- A **concurrent algorithm** (or **concurrent program**) is a formal description of the behavior of *a set of sequential state machines* that cooperate through a communication medium, e.g., a shared memory or by exchanging messages over a link
- Concurrent algorithm is a **multiprocess program**
  - Each process corresponding to the sequential execution of a given state machine

Process Synchronization

- **Process synchronization** occurs when the progress of one or several processes depends on the behavior of other processes
- Two types of process interaction require synchronization: **competition** and **cooperation**

Synchronization: Competition

- **Competition** occurs when processes compete to execute some statements (that access some shared resource) and only one process at a time (or a bounded number of them) is allowed to execute them
  - Example: when processes compete for a shared resource

Competition: Example

- Consider a disk with random (atomic) access I/O (low-level) block operations
  - seek(p) moves the disk head to location ‘p’
  - read() reads the contents of block at location ‘p’
  - write(v) writes ‘v’ in the block at location ‘p’ (overwriting previous contents)
- High-level process operations to be implemented/used:
  - disk_read(x) returns the contents of disk block ‘x’
  - disk_write(x,v) writes ‘v’ into disk block ‘x’

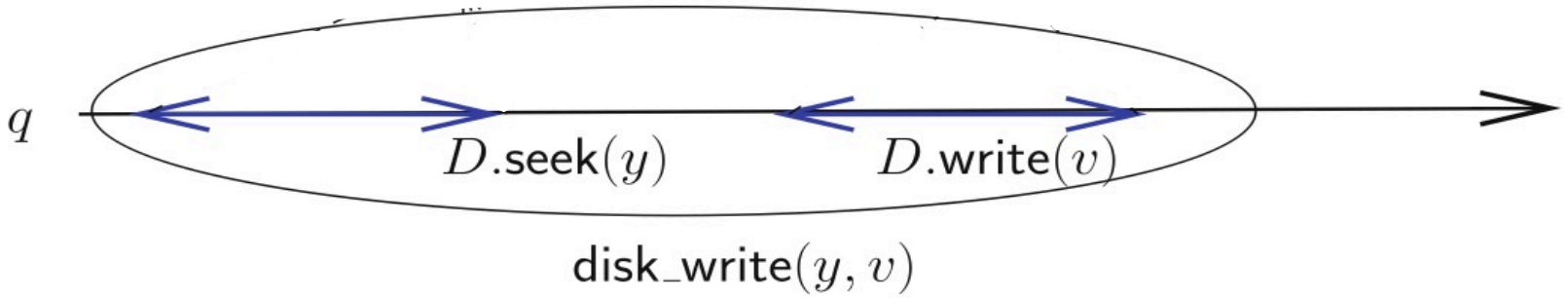
Operations on disk ‘D’

operation disk_read(x) **is**
    % r is a local variable of the invoking process %
    D.seek(x); $r \leftarrow D$.read(); return(r)
end operation.

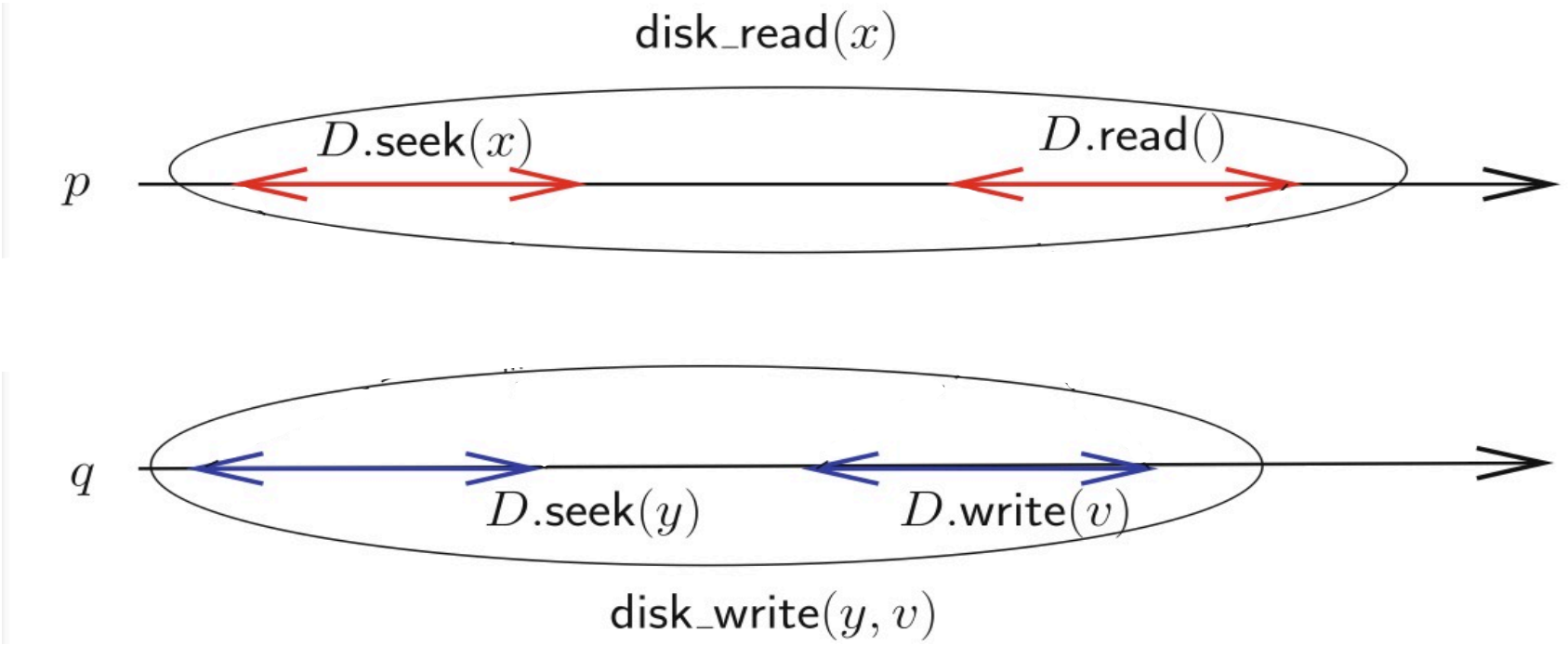
operation disk_write(x, v) **is**
    D.seek(x); D.write(v); return()
end operation.

An interleaving of disk operations

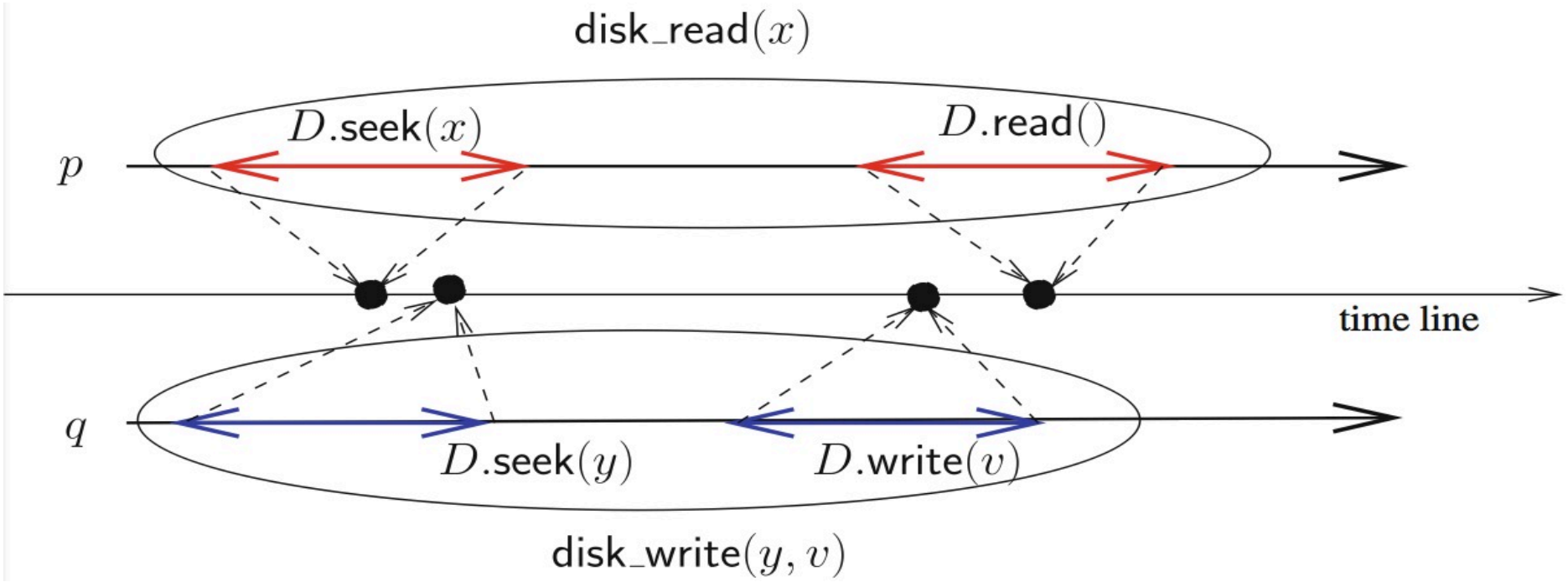
An interleaving of disk operations



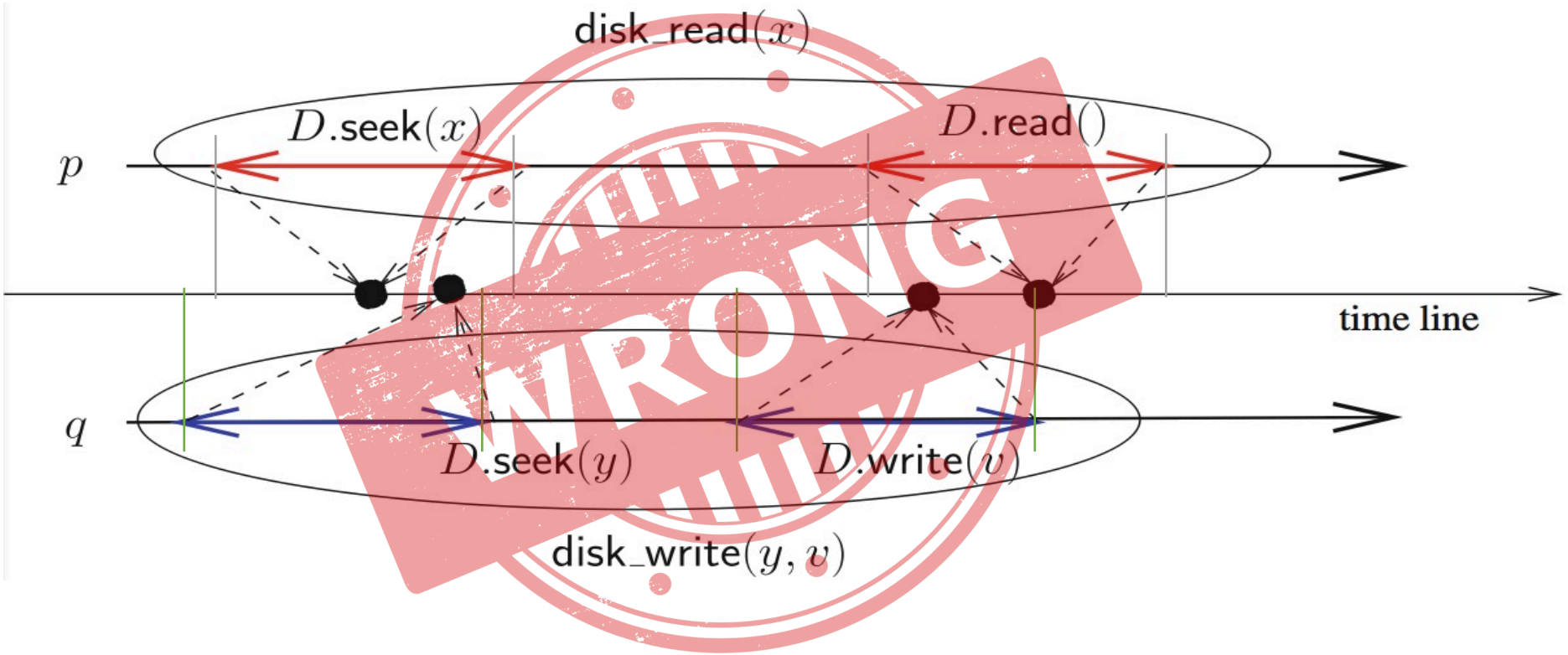
An interleaving of disk operations



An interleaving of disk operations



An interleaving of disk operations



Synchronization: Cooperation

- **Cooperation** occurs when one process can only progress after some event on another process
  - Example: when a processes waits for a signal sent by another process

Synchronization: Cooperation

- Barrier (or rendezvous)
  - A set of control points, one per process involved in the barrier, such that each process is allowed to pass its control point only when all other processes have attained their own control points
  - From an operational point of view, each process has to stop until all other processes have arrived at their control point

Synchronization: Cooperation

- A producer–consumer problem
  - The producer loops forever on producing data items
  - The consumer loops forever on consuming data items
- The problem consists in ensuring that
  a) only data items that were produced are consumed, and
  b) each data item that was produced is consumed exactly once
- How to solve this problem?

Solutions to the prod–cons problem

- Use a synchronization barrier
  - Both the producer (when it has produced a new data item) and the consumer (when it wants to consume a new data item) invoke the barrier operation
  - When, they have both attained their control point, the producer gives the data item it has just produced to the consumer
- This coordination pattern works but…
  - …it is not very efficient (overly synchronized): for each data item, the first process that arrives at its control point must wait for the other process

Solutions to the prod–cons problem

- Use a shared buffer of size $n \geq 1$
  - I.e., a queue or a circular array
  - The producer adds new data items to the end of the queue
  - The consumer withdraws the data item at the head of the queue
- Properties of a prod-cons with a buffer of size *n*:
  - A producer must wait only when the buffer is full
    - It contains *n* data items produced and not yet consumed
  - A consumer must wait only when the buffer is empty
    - Occurs each time all data items that have been produced have been consumed

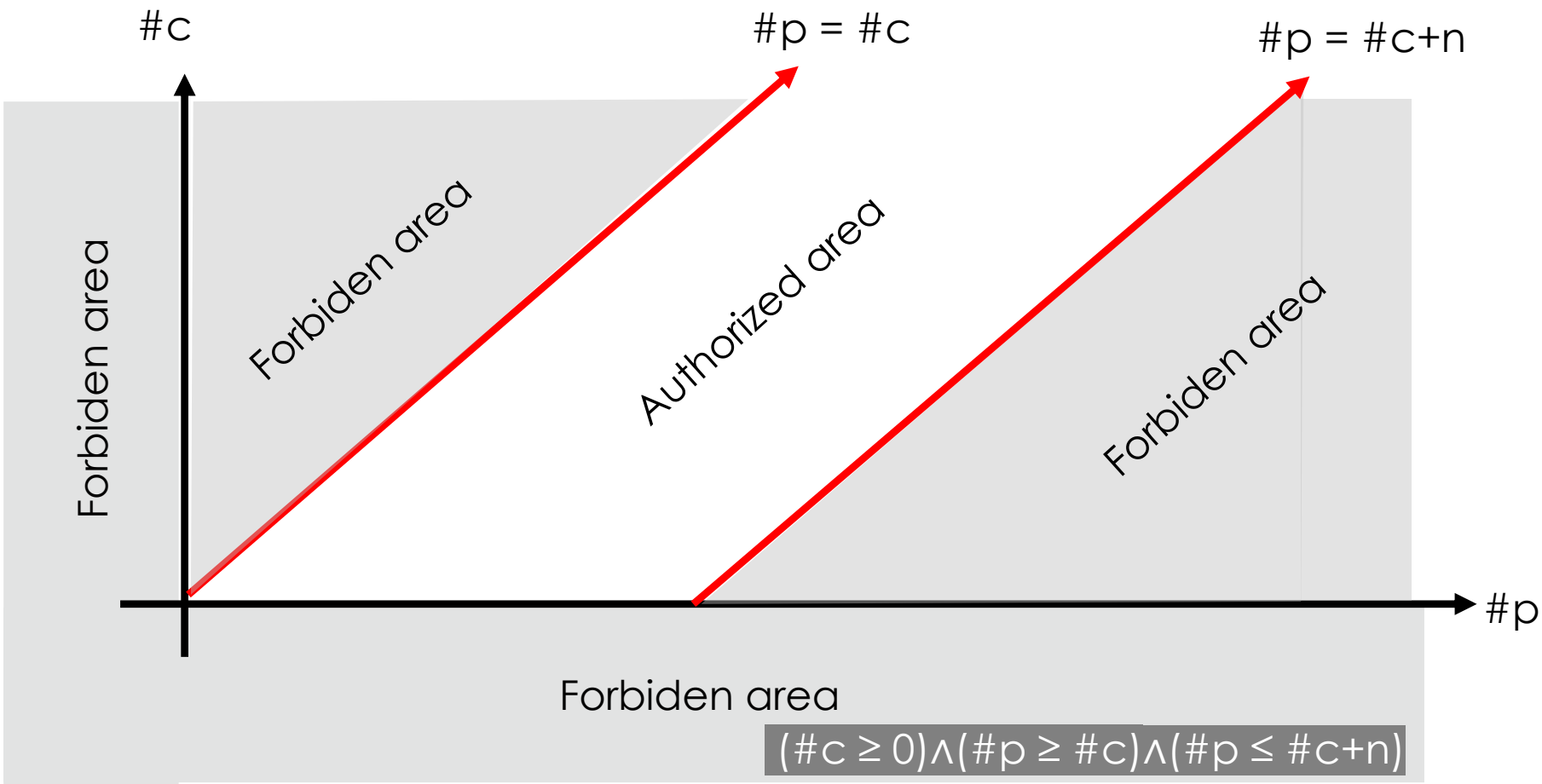
Synchronization: Invariants

- The **aim** of **synchronization** is to **preserve invariants**

- #p = number of data items produced so far
- #c = number of data items consumed so far

Synchronization: Invariants

#c = number of data items consumed so far
#p = number of data items produced so far



Synchronization: Invariants

- The **aim** of **synchronization** is to **preserve invariants**

- #p = number of data items produced so far
- #c = number of data items consumed so far
- Invariant for a buffer of size *n* is

$$(\#c \geq 0) \wedge (\#p \geq \#c) \wedge (\#p \leq \#c+n)$$

The Mutual Exclusion Problem

- Critical section
  - Part of code *A* (i.e., an algorithm) or several parts of code *A*, *B*, *C*, … (i.e., different algorithms) that, for some consistency reasons, must be executed by a single process at a time
    - E.g., if a process is executing code *B*, no other process can be simultaneously executing the codes *A* or *B* or *C* or etc.

- The operations **disk_read()** and **disk_write()** from before were critical sections

Mutual exclusion

- How to provide the application processes with an appropriate abstraction level?
- Designing
  - an **entry algorithm** (also called **entry protocol**); and
  - an **exit algorithm** (also called **exit protocol**)
  - that, when used to **delimit a critical section** *cs_code(in)*, **ensure** that the **critical section code is executed** by at most **one process at a time**
- Operations:
  - **acquire_mutex()** and **release_mutex()**

Mutual exclusion: concurrent execs of acquire_mutex()

- Only one of the invocations terminates
  - The corresponding process ***p*** is called the winner
- The other invocations stay in hold
  - The competing processes q_i are the losers
    - Their invocations remain pending
- A well formed process executes the entry and exit protocols appropriately

```
procedure protected_code(in) is
    acquire_mutex(); r ← cs_code(in); release_mutex(); return(r)
end procedure.
```

Mutual exclusion: definition

- The mutual exclusion problem consists in implementing the operations *acquire_mutex()* and *release_mutex()* in such a way that the following properties are always satisfied:
- **Mutual exclusion**, i.e., at most one process at a time executes the critical section code
- **Starvation-freedom**, i.e., for any process ‘p’, each invocation of *acquire_mutex()* by ‘p’ eventually terminates

Mutual exclusion: properties

- Safety
  - Safety properties state that nothing bad happens
  - They can usually be expressed as invariants
    - The invariant here is the mutual exclusion property, which states that **at most one process at a time** can execute the critical section code
    - *Note that a solution in which no process is ever allowed to execute the critical section code would trivially satisfy the safety property*
- Example of safety property:
  - Deadlock-freedom
    - Whatever the time τ, if before τ one or several processes have invoked the operation acquire_mutex() and none of them has terminated its invocation at time τ, then there is a time τ'> τ at which a process that has invoked acquire_mutex() terminates its invocation

Mutual exclusion: properties

- Liveness
  - Liveness properties state that something good eventually happens
- Example of liveness property:
  - Starvation freedom
    - Means that a process that wants to enter the critical section can be bypassed an arbitrary but finite number of times by each other process

Mutual exclusion: properties

- Starvation-freedom implies deadlock-freedom
  - If a process requests access to the critical section it will eventually get permission
  - To get permission the system cannot be deadlocked
- Deadlock-freedom does not imply starvation-freedom
  - The system is operating
  - There is a process willing to get access to the critical section that is always overcome by another later process

The Lock Object

- A lock is a shared object with two methods:
  - **LOCK.acquire_lock()** and **LOCK.release_lock()**
- A lock can be in one of two sates:
  - **free** or **locked**
- And is initialized
  - to the value **free**
- Its behavior is defined by a sequential specification
  - from an external observer point of view, all the **acquire_lock()** and **release_lock()** invocations appear as if they have been invoked one after the other
  - Sequence: `(LOCK.acquire_lock(); LOCK.release_lock())*`

The END